المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY

1ST YEAR BASIC TRAINING IN CYBER SECURITY

INTRODUCTION TO OPERATING SYSTEMS 1 (SYST1)

Dr. Sassi BENTRAD
✉ : sassi.bentrad.enscs@gmail.com | | sassi.bentrad@enscs.edu.dz
LISCO Laboratory (**L**aboratoire d'**I**ngénierie des **S**ystèmes **CO**mplexes) / University of Badji Mokhtar-Annaba (**UBMA**)
National **S**chool of **C**yber **S**ecurity (**NSCS**)

Basic Training in Cyber Security (1BT)

Formation de Base en Cyber-Sécurité (1FB)

Chapter 6

Redirection, Pipes and Filters

SYST1'2025/2026

Module Description Sheet : *Introduction to Operating Systems 1 (SYST1)*

Course Content

Redirection, Pipes and Filters (15 %)

CHAPTER 6

- Introduction
- Redirection
  - ✓ *Inputs and Outputs (I/O); Standard Input, Output and Error Output*
  - ✓ *Redirection symbols ; Reading from a file*
  - ✓ *Progressive reading from the keyboard ; Error Redirection*
  - ✓ *Merging of outputs ; Combining Redirections*
- Pipes
  - ✓ *Pipe syntax*
  - ✓ *Connecting more than two commands*
  - ✓ *Commands grouping (or chaining)*
  - ✓ *Imbrication of commands*
- Filters
  - ✓ *Introduction*
  - ✓ *Filter categories*
  - ✓ *Data processing commands (extract, sort, and filter)*

In everyday **Unix/Linux usage**, the combination of **REDIRECTION**, **PIPES**, and **FILTERS** provides a flexible and powerful mechanism for efficiently performing complex data manipulation tasks.

❑ Objectives

By the end of this chapter, students will :

- Understand the concepts of **input/output (I/O) redirection** and how to use it effectively.
- Master the use of **pipes** to connect commands and process data efficiently.
- Efficiently manipulate text data using **filters**' tools.
- Gain hands-on experience through practical examples and exercises.

❑ Prerequisites

Before starting this chapter, students should :

- Have **basic familiarity with the Linux command line**.
  - ✓ Know how to **navigate the file system** (cd, ls, pwd).
  - ✓ Understand **basic file operations** (cp, mv, rm, mkdir).
  - ✓ Be comfortable running commands and interpreting their output.
- Have access to a Linux environment ...

❑ Importance of Redirection, Pipes, and Filters

- **Redirection :**
  - ✓ Allows users to control where the input and output of commands go.
  - ✓ Essential for managing data flow, saving output to files, and handling errors.

- **Pipes :**
  - ✓ Enable the chaining of commands, where the output of one command becomes the input of another.
  - ✓ Facilitates complex data processing and filtering.

- **Filters :**
  - ✓ Data Processing and Transformation: make it easy to extract, manipulate, and format data
  - ✓ Powerful Text Manipulation: automate complex text replacements, data formatting, and pattern matching
  - ✓ Reducing Redundancy: simplify repetitive tasks

❑ Real-World Applications

REDIRECTIONS, **PIPES**, and **FILTERS** are essential tools in the **Linux/Unix shell environment**. They allow you to manipulate data in a powerful and efficient way. In real-world applications, these features are commonly used to automate tasks, manage system processes, and streamline data processing.

How they are applied in various real-world scenarios ?

- **Redirection:**
  - ✓ Saving command output to a file for later analysis.
  - ✓ Logging errors to a separate file for debugging.
- **Pipes:**
  - ✓ Filtering and processing log files.
  - ✓ Combining multiple commands to generate reports.
- **Filters:**
  - ✓ Searching and Filtering Data from Remote Systems
  - ✓ Using Filters for Text Processing

❑ Real-World Applications

REDIRECTIONS, **PIPES**, and **FILTERS** are foundational concepts, enabling efficient data processing, automation, and system management. Below are practical applications across various domains:

1. **Log management**: Redirecting logs to files for later analysis and monitoring.
2. **System resource monitoring**: Sorting and filtering processes to track resource usage.
3. **Automation of backups**: Combining search, backup & data migration , and compression commands for efficient data protection.
4. **Automation Scripts :** Ensures reliable backups with error tracking.
5. **Data processing**: Filtering, transforming, and processing large datasets, such as CSV files, for reporting and analysis.
6. **Remote system administration**: Using SSH, pipes, and filters to execute commands and collect results from remote servers.
7. **Real-time monitoring and logging**: Simultaneously viewing and saving important system outputs.
8. **Security Auditing :** Highlights potential security threats for action.

❑ Key Benefits

- **Efficiency**: Process large datasets without intermediate files.
- **Automation**: Combine commands in scripts for repeatable workflows.
- **Modularity**: Use small, specialized tools for complex tasks.
- **Scalability**: Handle streaming data or batch jobs seamlessly.

By mastering **REDIRECTION**, **PIPES**, and **FILTERS**, users can build powerful workflows for system administration, software development, and data analysis —perfectly illustrating the Unix philosophy of "***doing one thing well***."

Module Description Sheet : *Introduction to Operating Systems 1 (SYST1)*

Course Content

Input/Output Redirection (5 %)

PART 1

- Input and Output Streams
  - Concept of **Input/Output (I/O)**
  - Standard Input **(stdin)** ; Standard Output **(stdout)** ; Standard Error **(stderr)**
  - **File descriptors** overview (0, 1, 2)
- Output Redirection
  - Redirecting standard output to a file (>, >>)
  - Overwriting vs appending output
- Input Redirection
  - Reading input from a file (<)
  - Progressive input reading from the keyboard
- Error Redirection
  - Redirecting standard error (2>)
  - Separating normal output from errors
  - Error logging techniques
- Combining and Merging Redirections
  - Merging standard output and error (2>&1)
  - Redirecting multiple streams simultaneously

❑ Inputs and Outputs (I/O)

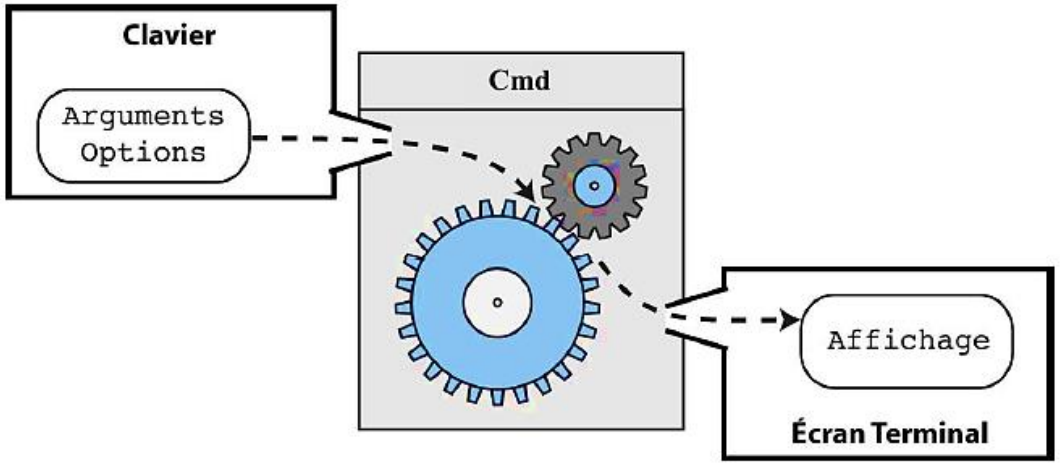
- **Objective:**

  Explain the basics of standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**) in Linux.

- **Topics Covered:**

  - What are **stdin**, **stdout**, and **stderr** ?
  - Default behavior of commands (***keyboard input***, ***terminal output***).
  - Examples of commands that use **stdin**, **stdout**, and **stderr**.

❑ Inputs and Outputs (I/O) : *Redirection*



How commands work ?

- Entering the command
- Display of the result

Possibility of redirecting the result :

- To a file
- As input to another command (sequence of commands)

Every **program** we run on the command line automatically has **three data streams** connected to it.

✓ A computer program takes **input data** (**input stream**).

✓ The program manipulates this **data**.

✓ The program provides an **output result** or **error** (**output stream**).

❑ Inputs and Outputs (I/O) : *Redirection*

- **Objective :**
  Learn how to redirect **input** and **output** in Linux.

- **Topics Covered :**

  - **Redirecting the result to a file (> and >>):**
    - Explanation of **>** (overwrite) and **>>** (append).
      Example: Redirecting command output to a file.

  - **Redirecting input (from a file or keyboard) (< and <<):**
    - Explanation of **<** (input from a file) and **<<** (here document).
      Example: Using a file as input for a command.

❑ Inputs and Outputs (I/O) : *Key Concepts to Remember*

- **Everything is a File:**
  - ✓ In Linux, input, output, and even devices are treated as **files**.
  - ✓ This philosophy makes **redirection** and **pipes** possible.
- **Standard Streams:**
  - ✓ **stdin** (**Standard Input**): The source of input data (default: keyboard).
  - ✓ **stdout** (**Standard Output**): The destination for command output (default: terminal).
  - ✓ **stderr** (**Standard Error**): The destination for error messages (default: terminal).
- **File Descriptors :**

  Numeric identifiers for input/output streams :
  - ✓ **0**: stdin
  - ✓ **1**: stdout
  - ✓ **2**: stderr

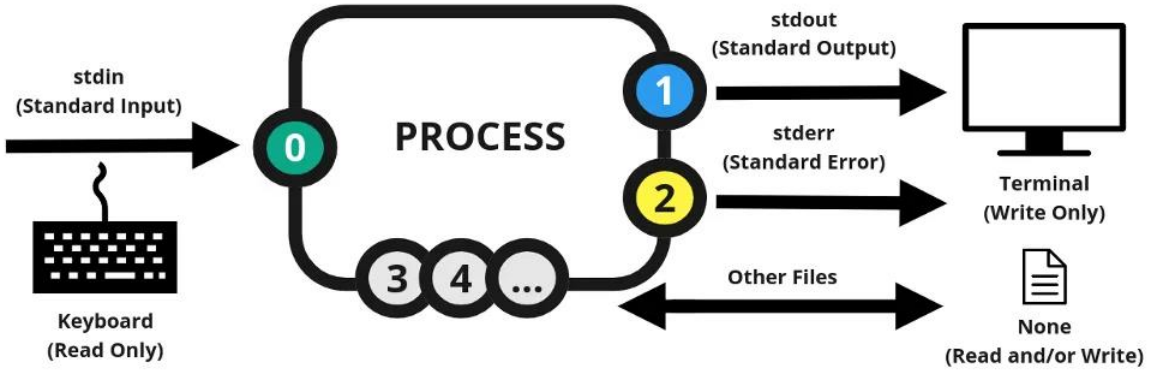
❑ Standard Input, Output and Error Output

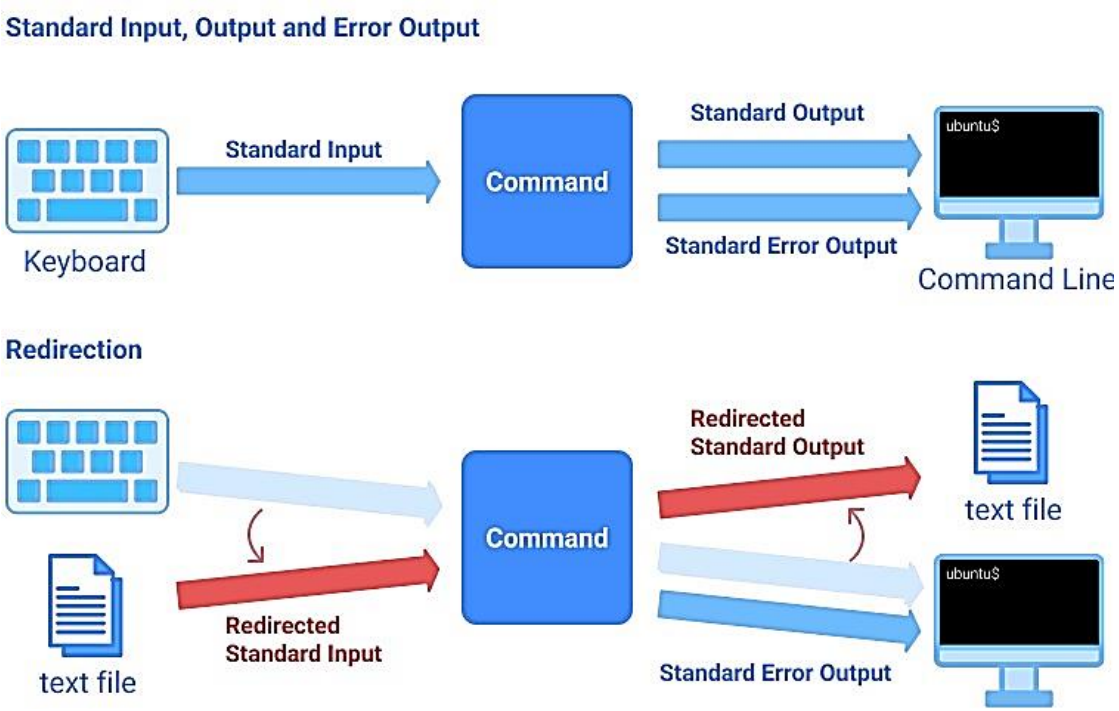
In general, computer commands or programs process **input** and generate **output**. **Standard input** and **output** define where data come from and where processed data go to.

Important concept :

Streams

Every program you may run on the command line has **3 streams**, **STDIN**, **STDOUT** and **STDERR**.





LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

❑ Standard Input, Output and Error Output

Three types of data flows can be distinguished :

- **Standard Input (STDIN) [0] :** *default place from which programs read*
  Standard input defines where input data comes from. Linux OS's preset standard input is generally the **keyboard**. When you type in the command line, the typed string becomes the input of the command.

- **Standard Output (STDOUT) [1] :** *default place to which programs write*
  Standard output defines where output data are generated. Linux OS's preset standard output is generally command-line (**terminal**).

- **Standard Error (STDERR) [2] :** *default place where errors are reported*
  Linux OS's preset standard error output is generally command-line (**terminal**).

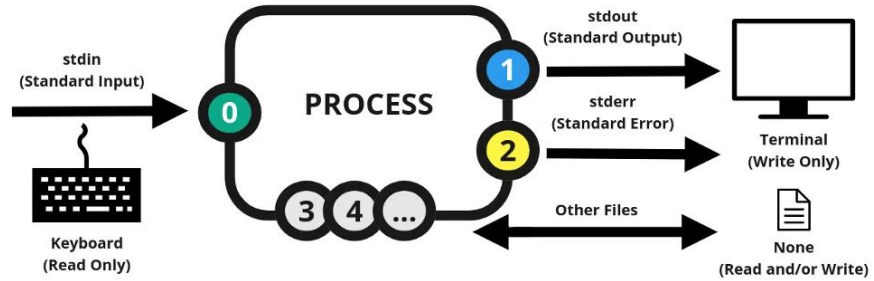
*Each one is identified by a **descriptor number**.*

Important concept :

Streams

Every program you may run on the command line has **3 streams**, **STDIN**, **STDOUT** and **STDERR**.

Standard Streams	Default
Standard Input (Stdin)	Keyboard
Standard Output (Stdout)	Terminal Display
Standard Error (Stderr)	Terminal Display



❑ Standard Input, Output and Error Output

In computing, these **three standard streams** are use programs to interact with the operating system and the user. Each one is associated with a **File Descriptor (FD)**, which is a unique identifier used to access the stream.

- **Standard Input (stdin):**
  - ✓ File Descriptor: **0**
  - ✓ Purpose: Used to read input data, typically from the keyboard or another input source.
  - ✓ Example: When a program reads user input, it uses stdin.
- **Standard Output (stdout):**
  - ✓ File Descriptor: **1**
  - ✓ Purpose: Used to write output data, typically displayed on the screen or sent to another output destination.
  - ✓ Example: When a program prints results to the console, it uses stdout.
- **Standard Error (stderr):**
  - ✓ File Descriptor: **2**
  - ✓ Purpose: Used to write error messages or diagnostics, typically displayed on the screen or logged to a file.
  - ✓ Example: When a program encounters an error and needs to report it, it uses stderr.

❑ Standard Input, Output and Error Output

In computing, these **three standard streams** are use programs to interact with the operating system and the user. Each one is associated with a **File Descriptor (FD)**, which is a unique identifier used to access the stream.

How They Work :

- These streams are automatically opened by the operating system when a program starts.

- By default :
  - **STDIN** is connected to the ***keyboard***.
  - **STDOUT** and **STDERR** are connected to the ***console*** (or ***terminal***).

- They can be **redirected to other sources or destinations** (e.g., files, pipes, or other programs).

❑ Redirection

Redirection is used to change **Stdin**, **Stdout,** or **Stderr**. In programming, the most frequently used redirection option is **redirection via a text file**.

Redirection of stdin (<)

To redirect standard input, use **<** after a command followed by new standard input.

```
command < file path for input
```

For example, if you want to sort the **file_a** contents, run the command below.

Command Line – INPUT

```
ubuntu $ | sort < file_a
```

❑ Redirection

Redirection of stdin (<) *Redirecting Standard Input*

- `cat < oldfile > newfile`

- **A more useful example:**
  - `tr string1 string2`
    - ✓ Read from standard input.
    - ✓ Character *n* of **string1** translated to character *n* of **string2**.
    - ✓ Results written to standard output.

  - **Example of use:**

    `tr a-z A-Z < file1 > file2`

❑ Redirection

Redirection of stdout (> or >>)

To redirect standard output, use > or >> after a command followed by a new standard output.

When you use >, the output will overwrite the existing contents while when you use >>, the output will be added at the end of the existing contents.

Overwrite contents : `command > file path for output`
Append contents : `command >> file path for output`

For **example**, run the following command to save the output of the **ls** command for the **/** (**root**) directory in the file **`list.txt`**. To avoid a permission error, run the command as the superuser.

❑ Redirection

Redirection of stdout (> or >>) *Redirecting Standard Output*

- **`cat file1 file2 > file3`**
  - ✓ concatenates file1 and file2 into file3
  - ✓ file3 is created if not there
- **`cat file1 file2 >! file3`**
  - ✓ file3 is clobbered if there
- **`cat file1 file2 >> file3`**
  - ✓ file3 is created if not there
  - ✓ file3 is appended to if it is there
- **`cat > file3`**
  - ✓ file3 is created from whatever user provides from standard input

❑ Redirection

Redirection of stderr (2> or 2>>)

To redirect a standard error, use **2>** or **2>>** after a command followed by a new standard error.

Overwrite contents : `command 2> file path for error output`
Append contents : `command 2>> file path for error output`

When you use **2>**, the output will overwrite the existing contents while when you use **2>>**, the output will be added at the end of the existing contents.

To see how it works, run the command below as a normal user. As the normal user doesn't have the read permission to the `/lost+found` directory, you'll get an error message.

❑ Redirection

Redirection of stderr (2> or 2>>) *Redirecting Standard Error*

- Generally direct standard output and standard error to the same place:

```
ubuntu[1] > cat myfile >& yourfile
```

  - ✓ If myfile exists, it is copied into yourfile
  - ✓ If myfile does not exist, an error message
    cat myfile : No such file or directory is copied in yourfile

- In **tcsh (TENEX C Shell)**, to write standard output and standard error into different files:

```
ubuntu[2] > (cat myfile > yourfile) >& yourerrorfile
```

- In **sh (Bourne Shell)**, standard error is redirected differently

```
cat myfile > yourfile 2> yourerrorfile
```

❑ Redirection

Redirection of stdout and stderr in the same file

If you want to record standard output and error in the same file, use **>&** symbol.

```
command >& file path for output and error
```

Redirection of stderr (2> or 2>>)

To redirect a standard error, use **2>** or **2>>** after a command followed by a new standard error.

When you use **2>**, the output will overwrite the existing contents, while when you use **>>**, the output will be added at the end of the existing contents.

Redirection of stdin and stdout with the same command

You can redirect both standard input and output with the same command.

```
command < file path for input > file path for output
```

❑ Redirection symbols

Here is the summary of **redirection symbols**.

Symbol	Content
>	set **standard output** (**overwrites** the data of the file when the same file name exists)
>>	set **standard output** (**appends** the data of the file when the same file name exists)
<	set **standard input**
2>	set **standard error** (**overwrites** the data of the file when the same file name exists)
2>>	set **standard error** (**appends** the data of the file when the same file name exists)
>&	set **standard output and standard error** to the same file

There are **three redirectors** to work with: **>**, **>>**, and **<**. The following information describes each one:

Redirection with >

- **command > file**: Sends standard output to <file>
- **command 2> file**: Sends error output to <file>
- **command 2>&1**: Sends error output to standard output
- **command > file 2>&1**: Sends standard output and the error output to a file
- **command &> file**: Sends standard output and the error output to a file
- **command 2>&1 > file**: Sends error output to standard input and the standard input to a file

Append with >>

- **command >> file**: Appends standard output to a file
- **command 2>> file**: Appends error output to a file
- **command >> file 2>&1**: Appends standard output and error output to a file
- **command &>> file**: Appends standard output and error output to a file
- **command 2>&1 >> file**: Sends error output to standard input and appends standard input to a file

Redirect with <

- **command < input**: Feeds a command input from <input>
- **command << input**: Feeds a command or interactive program with a list defined by a delimiter; this is known as a here-document (heredoc)
- **command <<< input**: Feeds a command with <input>; this is known as a here-string

❑ Reading from a file

Consists of recovering the typed data from a file. `cmd < fichier.in`
Not used much because most commands accept a filename as an argument.

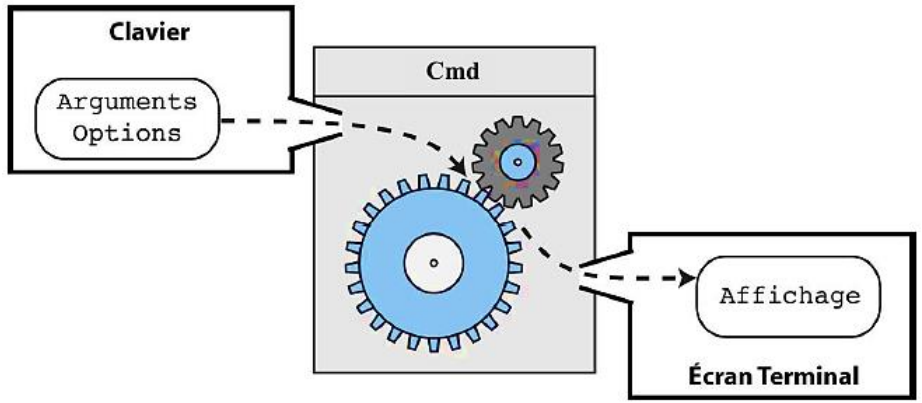
Example 01 :

```
sort ListeSE.txt
sort < ListeSE.txt
```

Example 02 :

```
wc -| ListeSE.txt
60 ListeSE txt

wc -| < ListeSE.txt
60
```



✓ The command takes the file name as input and then opens it.

✓ The command receives the contents of the file, (sent by the Shell).

❑ Reading from a file

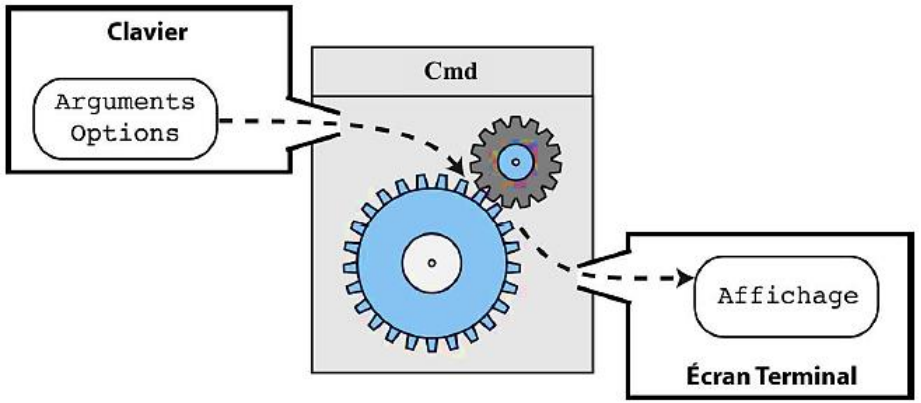
Consists of recovering the typed data from a file. `cmd < fichier.in`
Not used much because most commands accept a filename as an argument.

Example 03 :

The command **tr** is used to convert or delete characters.

```
tr abc ABC
```

- afoce → AfBCe
- Salut → sAlut
- ataax → AtAAX *(**Ctrl + D** to stop typing)*



Example 04 :

- A file ListeSE.txt : afbce Salut ataax

```
tr abcd ABCD < ListeSE.txt
```

AfBCe SAlut AtAAx

❑ Progressive reading from the keyboard <<

- Allows you to send content to a command using the keyboard.
- Avoid creating a file you don't need.

Example 01 :

```
sort ListeSE.txt
sort << END
```

*Enter the items to be sorted using the keyboard, then **END** to indicate the end of the entry.*

```
user1@PC:~$ sort -n << END
> 120
> 10
> 1
> 30
> END
1
10
30
120
```

Example 02 :

```
wc -m ListeSE.txt
wc -m << END
```

*Enter the elements using the keyboard, then **END** to indicate the end of the entry.*

```
user1@PC:~$ wc < fichier
  8  24 208
```

*Is it possible to enter something other than **END** to indicate the end of the entry ?*

❑ Progressive reading from the keyboard <<

- Allows you to send content to a command using the keyboard.
- Avoid creating a file you don't need.

Example 03 :

```
tr abcd ABCD << END
> aaatttbbb
> bbbfffzzz
> END
AAAtttBBB
BBBfffzzz
```

Example 04 :

```
tr abcd ABCD << END
> aaatttbbb
> END
AAAtttBBB
```

END could be substituted by **EOF**, **STOP**

❑ Error Redirection

Commands produce two different streams:

- **Standard output**: for all messages (except errors).
- **Error output**: for all errors.

- By default, everything is displayed on the screen.
- **Redirection:** Consists of returning error messages to a file.

✓ **Overwriting the contents of the file:** `cmd 2> fichier.err`

- **Example 01**: `ls Doc/ 2> ListeErreur.txt`
- Creating or overwriting the file **`ListeErreur.txt`**

✓ **With preservation of the file's contents:** `cmd 2>> fichier.err`

- **Example 02**: `ls Doc/ 2>> ListeErreur.txt`
- Create or add to the end of the file **`ListeErreur.txt`**

❑ Merging of outputs

Consists of merging the outputs (**standard output and errors**) into a **single file**.

```
command > output.log 2> error.log
```

Example 01:

```
ls Bureau/ Doc/ > output.log 2>&1
```

The **standard output** (regular listing of files) and **standard error** (any error messages) will both be written to `output.log`.

```
ls Bureau/ Doc/ 2> output.log >&2
```

The **standard output** is mixed with the **error output**.

❑ Combining Redirections

Objective: Show how to combine multiple redirections for advanced use cases. It is possible to make several redirects at once.

```
cmd > fichier.out 2> fichier.err
```

Example 01: `ls > ListeSE.txt 2> ListeErreur.txt`

List of items in the **`ListeSE.txt`** file and error messages in the file **`ListeErreur.txt`**

```
cmd < fichier.in > fichier.out
```

Example 02: `tr abc ABC < ListeSE.txt > ListeSE_New.txt`

- Reading from the file **`fileListeSE.txt.`**
- Replaces occurrences of characters a, b, and c in the file **`ListeSE.txt`** with A, B, and C.
- Create a file named **`ListeSE_New.txt`** that contains the new text after editing.

❑ /dev/null ... ?

- A virtual file that is **always empty**.
- Copy things to here and they disappear : `cp myfile /dev/null`
- Copy from here and get an empty file : `cp /dev/null myfile`
- Redirect error messages to this file : `(ls -l > recordfile) >& /dev/null`
- Basically, all error messages are discarded.

Why Use `/dev/null` ?

✓ It's useful when you want to run a command for its side effects (e.g., *triggering a script*) but don't need to see its output.

✓ You can also use `/dev/null` in scripts where you need to redirect unwanted output away from the terminal or log files.

`/dev/null` is a place to "**throw away**" data, whether it's standard output or error messages. It's a handy tool for cleaning up or silencing commands when you don't want their output cluttering your terminal or logs.

Module Description Sheet : *Introduction to Operating Systems 1 (SYST1)*

Course Content

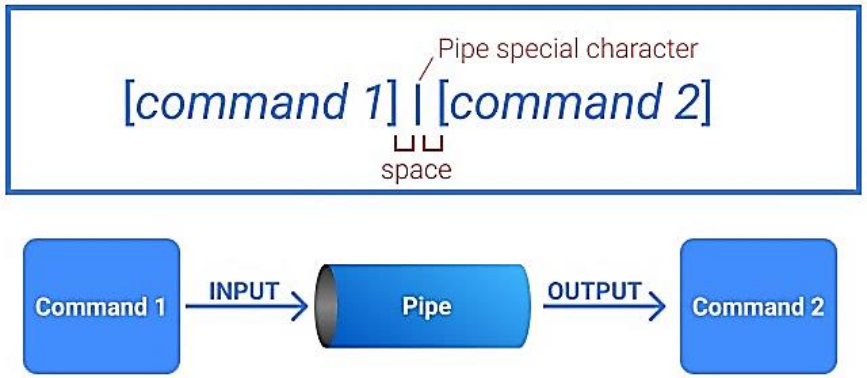
Pipes (5 %)

PART 2

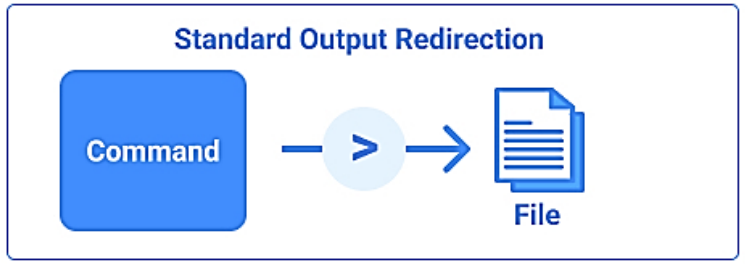
- Pipes
  - Concept of pipes (|)
  - Data flow between processes
  - Standard input and standard output (stdin/stdout)
- Pipe Syntax
  - Basic pipe syntax and usage ; Common examples (ls | grep, cat | wc)
- Connecting More Than Two Commands
  - Multi-stage pipelines
  - Linear command chains
- Command Grouping (Chaining)
  - Command grouping with parentheses ( )
  - Command chaining with && and ||
  - Sequential execution vs conditional execution
- Command Nesting (Imbrication)
  - Nested commands using command substitution
  - Combining nesting with pipes
  - Practical examples and best practices

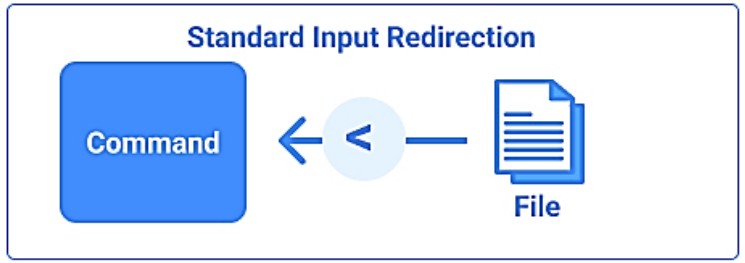
❖ PIPES

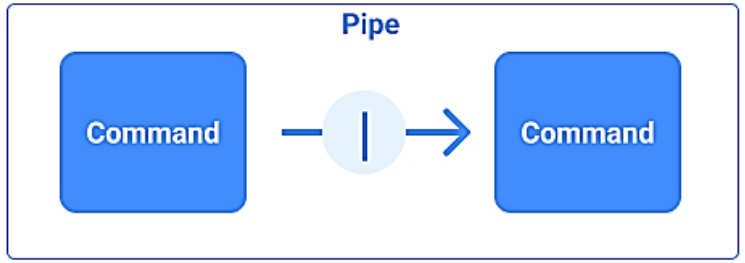
Pipe is used to combine two or more commands. **Outputs** **of the first command become inputs** **of the second command.**



LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

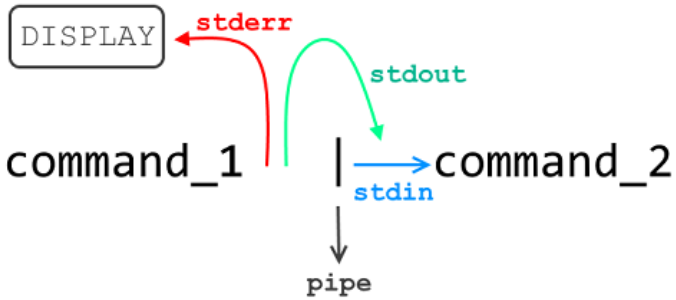






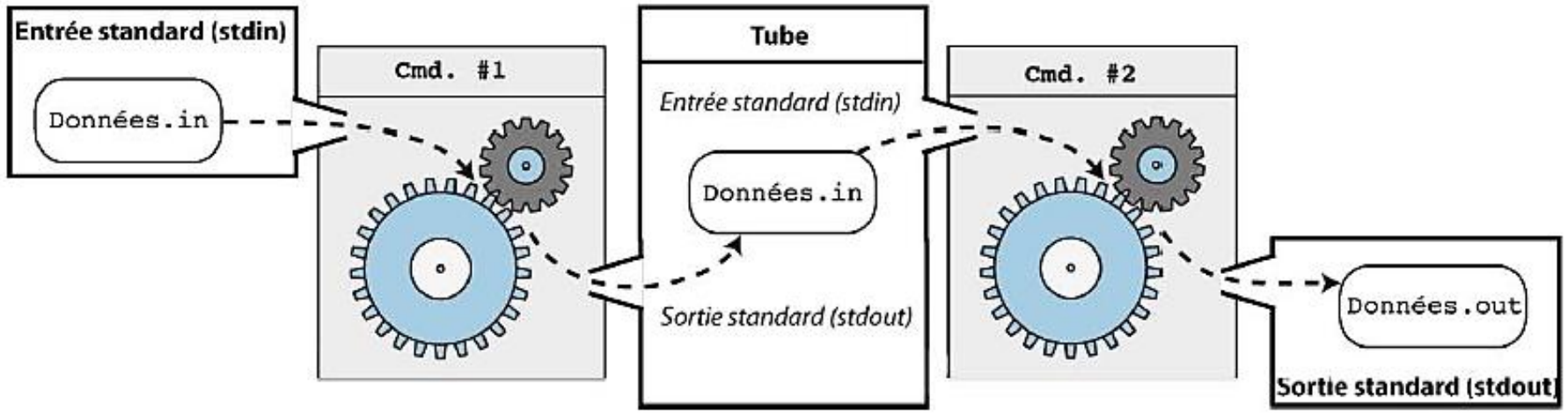
Difference between Redirection and Pipe :

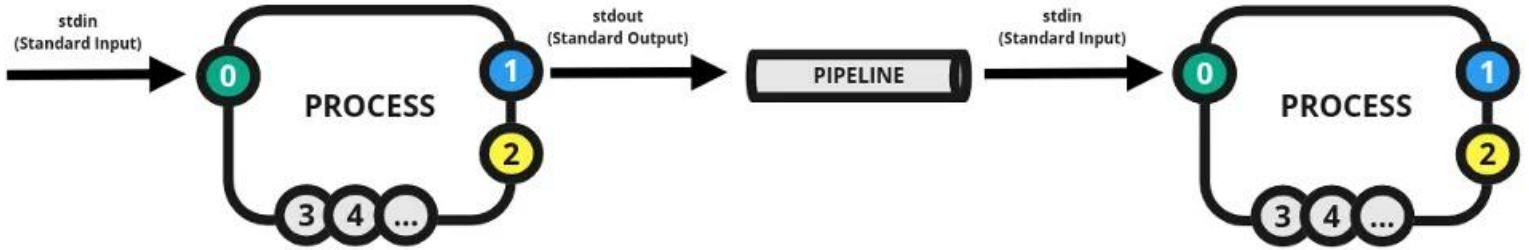
Redirection is used mostly to connect a **command with a file** while the **pipe** is used to **connect commands** as shown in the illustration below.



❖ PIPES

In short, the output of each process directly as input to the next one like a **pipeline**. We can use powerful commands which can **perform complex tasks in an instant**.





❑ Pipe syntax

Using pipes is simple - you just need to connect two commands with the **pipe** symbol " | ".
To demonstrate how the pipe works with a simple example, combine the **ls** command and the **grep** command to display a list of directories under */ (root)* that contain a certain character in their name.

First, run the **ls** command without the pipe to see the difference.

Then, run the **ls** command while piping the **grep** command to list only the directories with r in their name.

You'll see a result like the one below.

Command Line- INPUT

```
ubuntu $ | ls /
```

Command Line- RESPONSE

```
bin dev home lib32 libx32      media opt  root sbin srv
tmp var
boot etc lib lib64 lost+found  mnt   proc run  snap sys
usr
```

Command Line- INPUT

```
ubuntu $ | ls / | grep r
```

Command Line- RESPONSE

```
proc
root
run
srv
usr
var
```

❑ Connecting more than two commands

You can connect more than two commands with **pipes**.

Example :

```
user1@PC:~$ cat file
1 z
2 y
3 x
4 w
user1@PC:~$ cat file | sort -k 2
4 w
3 x
2 y
1 z
user1@PC:~$ cat file | sort -k 2 | grep -n w
1:4 w
```

❑ Commands grouping (or chaining)

- **Variables**

  Variables are symbols used to **temporarily** store data.

  To put a value in a **variable**, simply write:

```
variable_name=value
```

Attention :

- ✓ There must be **no space** around the symbol "**=**"
- ✓ The variables are **local** to the terminal
- ✓ Variables **disappear** when the terminal is closed

The **shell** replaces all variables **$variable_name** with their contents before a command is executed

```
user1@PC:~$ a=Hello
user1@PC:~$ Var1=123

user1@PC:~$ a= Hello
Hello : Command not found

user1@PC:~$ a =Hello
a : Command not found
```

```
user1@PC:~$ a=Hello
user1@PC:~$ echo $a
Hello

user1@PC:~$ path=../
user1@PC:~$ echo $path
../

user1@PC:~$ cd $path
user1@PC:/home$
```

❑ Commands grouping (or chaining)

- **$?**
  - ✓ **$?** is a variable that contains the execution status of the last command
    $? = 0 if there is no error
  - ✓ **$? ≠ 0** if there is an error

```
user1@PC:~$ pwd
/home/user1
user1@PC:~$ echo $?
0
user1@PC:~$ PWD
PWD: command not found
user1@PC:~$ echo $?
127
```

- **Semicolon ;**
  - ✓ Several commands executed independently on the same line.
  - ✓ **Syntax** : `cmd1 ; cmd2 ; cmd3 ;...`

```
user1@PC:~$ whoami hostname pwd
whoami: extra operand ‘hostname’
Try 'whoami --help' for more information.

user1@PC:~$ whoami ; hostname ; pwd
user1
PC
/home/user1
```

❑ Commands grouping (or chaining)

- **&&**
  - ✓ In the example above, execution continues even if an error occurs.
  - ✓ In order to stop, use **&&** (logical **AND**)
  - ✓ **Syntax** : `cmd1 && cmd2 && cmd3 && ...`
  - ✓ The second command is executed if the first succeeds.

```
user1@PC:~$ ls desktop/
TP1 TP2
user1@PC:~$ find desktop/ -name "TP*" && rm -f desktop/*
desktop/TP1
desktop/TP2
user1@PC:~$ ls desktop/
user1@PC:~$
```

- **||**
  - Alternatively, use **||** (logical **OR**)
  - In this case, the second command is executed if the first fails.
  - **Syntax** : `cmd1 || cmd2 || cmd3`

```
user1@PC:~$ PWD
PWD: command not found
user1@PC:~$ PWD || echo "There is an error"
PWD: command not found
There is an error
user1@PC:~$ pwd || echo "There is an error"
/home/user1
```

❑ Commands grouping (or chaining)

- Multiple commands are considered as one.
- Use **brackets ()**
- **Syntax :**

```
(cmd1; cmd2; cmd3)
(cmd1 && cmd2 && cmd3)
(cmd1 || cmd2 || cmd3)
```

❑ Imbrication of commands

- The command to be executed by its result.
- Use the result of one command as an argument to another.
- **Syntax :** `$(cmd)`
- **Examples :**

```
echo Current Directory : $(pwd)
echo Current Directory : $(pwd; who)
```

Module Description Sheet : *Introduction to Operating Systems 1 (SYST1)*

Course Content

Filters (5 %)

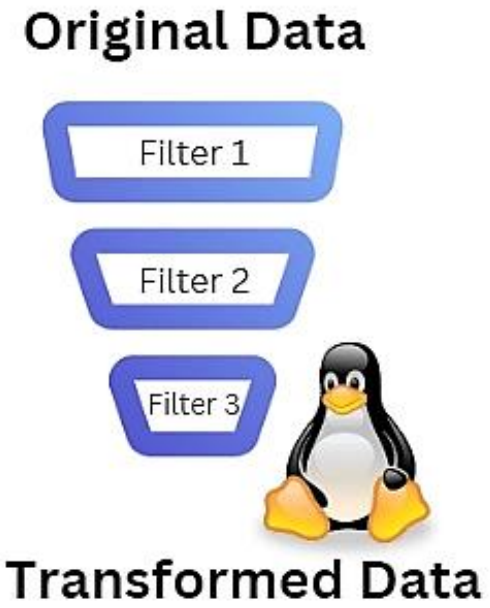
PART 3

- Counting
  - Counting lines, words, and characters ; The wc command
  - Typical use cases in administration and data analysis
- Selection
  - Selecting lines based on patterns or conditions ; Common tools: grep, sed
  - Regular expressions (basic overview)
- Classification
  - Sorting and ordering data ; The sort command
  - Numeric vs lexical sorting ; Removing duplicates with uniq
- Merging and Decomposition
  - Combining files and data streams ; Tools: paste, join
  - Splitting data into parts: cut, split ; Practical data preparation examples
- Comparison
  - Comparing files and streams ; Tools: diff, comm
  - Typical use cases (configuration files, logs)
- Character Transformation
  - Transforming characters and formats ; Use cases: case conversion, character replacement, ...

❑ What is a Filter ?

Filters are commands that read from standard input (**stdin**), modify or process the input data, and then send the result to standard output (**stdout**).

They allow you to transform data in a variety of ways, such as *formatting*, *searching*, *extracting*, or *replacing content*.



- Linux has a lot of filter commands like **awk**, **grep**, **sed**, **spell**, and **wc**.
- A **filter** takes input from one command, does some processing, and gives output.
- When you **pipe** two commands, the "**filtered** " output of the first command is given to the next.

❑ Importance

- **Data Processing and Transformation:** Filters make it easy to extract, manipulate, and format data.

  For example, **awk**, **sed**, **grep**, and **cut** are filters that let you handle everything from searching text files to processing structured data like CSVs.

- **Powerful Text Manipulation:** With filters like **sed** and **awk**, you can automate complex text replacements, data formatting, and pattern matching without needing to write custom scripts.

- **Reducing Redundancy:** Filters simplify repetitive tasks such as searching for specific patterns or performing transformations across multiple files. With just a few commands, you can avoid doing repetitive work manually.

❑ Example …

Let's understand this with the help of an example.
We have the following file '**sample**'

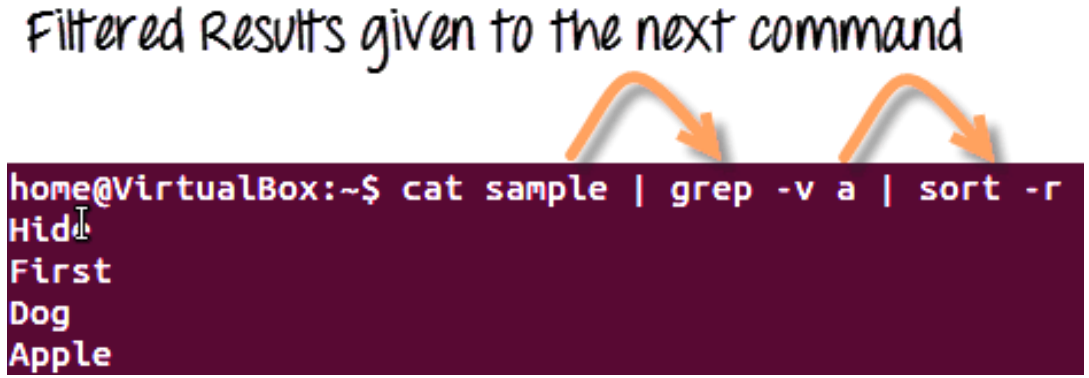
```
home@VirtualBox:~$ cat sample
Bat
Goat
Apple
Dog
First
Eat
Hide
```

We want to highlight only the lines that do not contain the character '**a**', but the result should be in reverse order.

For this, the following **syntax** can be used.

```
cat sample | grep -v a | sort - r
```

Let us look at the result.



❑ Filter categories

1. **Counting**
2. **Selection**
3. **Ranking**
4. **Melting and Decomposition**
5. **Comparison**
6. **Miscellaneous**: character transformation, etc.

❖ DATA PROCESSING COMMANDS (EXTRACT, SORT, AND FILTER) ...

Data processing in Linux involves **extracting**, **sorting**, and **filtering** data efficiently using **command-line tools**. Commands like **cut**, **awk**, and **sed** are commonly used to extract and manipulate data, while **sort** helps organize the data in ascending or descending order.

Filtering is achieved using tools like **grep** to search for patterns or exclude unwanted data. Combining these commands using **pipes** allows for the creation of powerful and flexible workflows, enabling quick and effective data manipulation directly from the terminal.

It's true that there are many **graphics programs** that are easy to use. However, the following commands are generally used by **system administrators** for its :

- **Efficiency**
- **Speed**
- **Power**

❑ Counting lines, words, and characters: wc

The **wc** (**word count**) command allows to count lines, words and characters.

Syntax : `wc [Options] [files]`

Some interesting options:

- **-l** counts the number of lines
- **- c** counts the number of bytes
- **- w** counts the number of words
- **- m** counts the number of characters

Example :

```
user1@PC:~$ wc /etc/passwd
50 79 2843 /etc/passwd

user1@PC:~$ wc -l /etc/passwd
50 /etc/passwd

user1@PC:~$ wc -w /etc/passwd
79 /etc/passwd

user1@PC:~$ wc -c /etc/passwd
2843 /etc/passwd
```

❑ Counting identical lines: uniq *(Extract Unique Data Lines)*

The **uniq** command allows you to remove successive duplicate rows from the standard entry or sorted files. With the **sort** command, you can extract unique data lines.

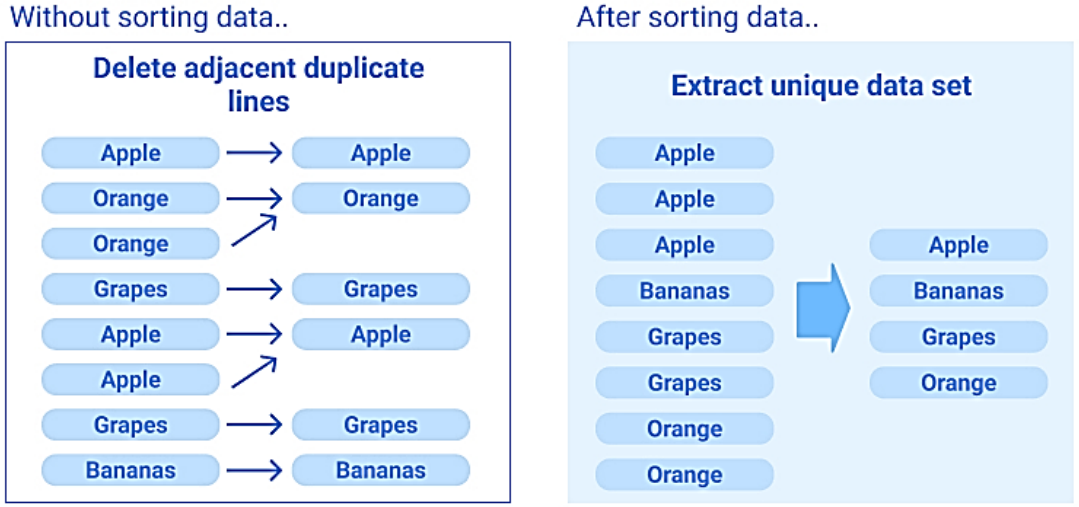
Syntax : `uniq [options][source[destination]]`

LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

Some interesting options :

- **`uniq`** : deletes identical (successive) lines
- **`uniq -c`** : counts successive identical lines
- **`uniq -d`** : displays duplicate lines only

`uniq` [*file path*]



❑ Counting identical lines: uniq *(Extract Unique Data Lines)*

Examples :

tp.txt	**$ uniq tp.txt**	**$ uniq -c tp.txt**	**$ uniq -d tp.txt**
cours :admin :3 td :user :2 tp :root :1 tp :root :1	cours :admin :3 td :user :2 tp :root :1	1 cours :admin :3 1 td :user :2 2 tp :root :1	tp :root :1

```
user1@PC:~$ cat test_uniq
omar
ali
omar
omar
omar
user1@PC:~$ uniq test_uniq
omar
ali
omar
```

❑ less (Display Content with Pager)

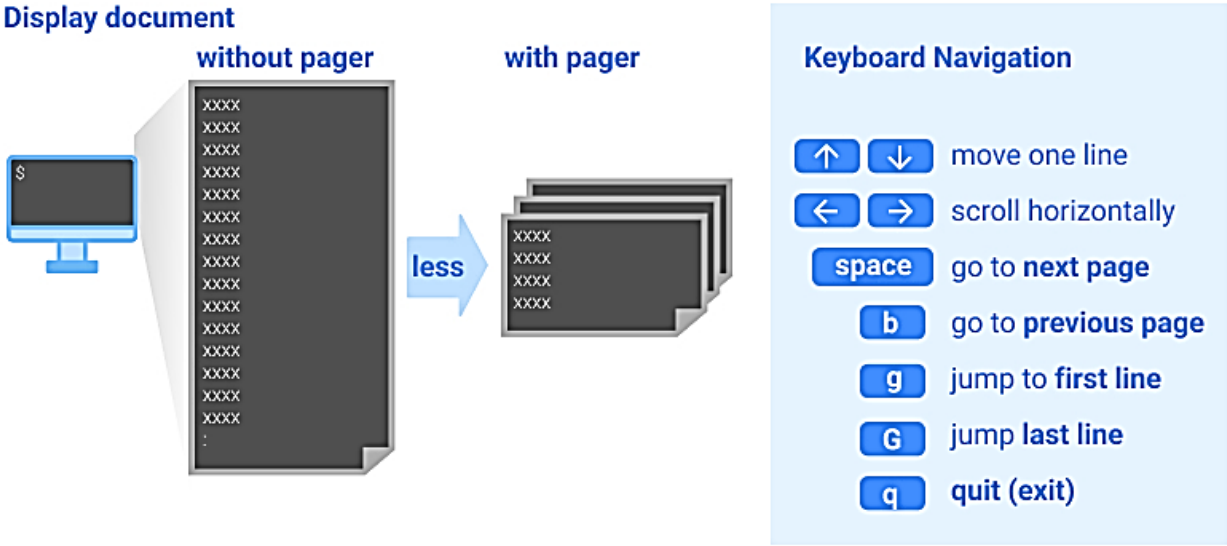
The **less** command is functioning as a **pager**, which is used to read the contents of a text file **one page** (one screen) at **a time**.

It is an efficient way to **read large log files** or **configuration files** without loading them entirely into memory.

With **less**, you can search, navigate, and even jump to specific lines, offering greater control over file viewing.

LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

```
less [file path]
```



❑ cut (Extract Data Sections)

The command **cut** is used to extract some data sections, which are usually called data fields.

This command is useful when the data set already has a consistent structure like data in a table format.

- By specifying the position (with **-c**), you can select columns of characters.
- By specifying a separator (with **-d**), you can select fields separated by a specific character.

LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

```
cut -f [field number] -d [delimiter] [file path]
```

❑ Selecting Columns: cut -c

Syntax : `cut –ccolumns [ files ]`

columns : Position of the characters to be extracted
Files : Files from which to extract

- A column is the position of a character in the line.
- The first character is column 1, the second column 2, etc.

Examples :

tp.txt	$ cut -c 2-5 tp.txt	$ cut -c -3 tp.txt	$ cut -c 4- tp.txt
tp :root :1	p :ro	tp :	root :1
tp :root :1	p :ro	tp :	root :1
td :user :2	d :us	td :	user :2
cours :admin :3	ours	cou	rs :admin :3

```
user1@PC:~$ cat test_cut
/bin/ls
/bin/mkdir
/bin/rm

user1@PC:~$ cut -c6,7,8,9,10 test_cut
ls
mkdir
rm

user1@PC:~$ cut -c6-10 test_cut
ls
mkdir
rm

user1@PC:~$ cut -c6- test_cut
ls
mkdir
rm
```

❑ Field selection: cut -d

Syntax :

```
cut –dseparator –ffields [fichiers]
```

Retains fields whose numbers are specified in the option **-f** (list or range)

- **-d**: specifies the separator in the file(s)
- **-f**: specifies the number of the field(s) to be cut

Examples :

tp.txt	$ cut -d : -f 1	$ cut -d : -f 1,3	$ cut -d : -f 2-
tp :root :1	tp	tp :1	root :1
tp :root :1	tp	tp :1	root :1
td :user :2	td	td :2	user :2
cours :admin :3	cours	cours :3	admin :3

```
user1@PC:~$ cat test_cut
/bin/ls
/bin/mkdir
/bin/rm

user1@PC:~$ cut -d"/" -f2,3 test_cut
bin/ls
bin/mkdir
bin/rm

user1@PC:~$ cut -d"/" -f2-3 test_cut
bin/ls
bin/mkdir
bin/rm

user1@PC:~$ cut -d"/" -f3 test_cut
ls
mkdir
rm
```

❑ Selecting lines from the beginning: head

The command **head** allows you to display only the first few lines of a file (or files)

Syntax : `head –n nbre[ files ]`

- `nbre :` Number of lines to display
- `files :` Files from which to extract

Examples :

tp.txt
```
tp :root :1
tp :root :1
td :user :2
cours :admin :3
```

$ head tp.txt
```
tp :root :1
tp :root :1
td :user :2
cours :admin :3
```

$ head –n 3 tp.txt
```
tp :root :1
tp :root :1
td :user :2
```

❑ Selecting lines by the end : tail

The command **tail** allows you to display the end of a file (or files)

Syntax : `tail -n nbre[ files ]`

- `nbre` : Number of lines to display
- `files` : Files from which to extract

Examples :

tp.txt	**$ tail tp.txt**	**$ tail -n 3 tp.txt**	**$ tail -n +2 tp.txt**
tp :root :1	tp :root :1		
tp :root :1	tp :root :1	tp :root :1	tp :root :1
td :user :2	td :user :2	td :user :2	td :user :2
cours :admin :3	cours :admin :3	cours :admin :3	cours :admin :3

❑ Selection of common lines: comm

The comm command is used to extract the common lines to two sorted files.

Syntax : `comm file1 file2`

Examples :

tp1.txt
td :user :1
td :usr1 :3
tp :admin :4
tp :root :2

tp2.txt
td :usr2 :1
td :usr3 :3
tp :admin :4
tp :root :2

$ comm tp1.txt tp2.txt
tp :root :2
tp :admin :4

❑ Sorting lines in a File : sort

The command **sort** allows you to sort lines, by default in ascending lexicographic order.

Syntax : `sort [option] [files]`

Some interesting options:

- **-k** to specify the column number used for sorting
- **- o** to write the result to a file
- **- r** reverse sort (descending order)
- **- n** numeric sorting, ideal for columns of numbers

Note :

✓ Sorting is possible on one or more fields. The default field separator is **tab** or **at least one space**.

✓ If there are multiple spaces, the **first one is a separator** and the other ones are characters that are part of the next field.

❑ Sorting lines: sort

Examples :

tp.txt	$ sort -t : -k 2 -n	$ sort -t : -k 4n,4 -k 2n,2
tp :2 :root :1	td :1 :user :2	tp :2 :root :1
tp :4 :root :1	tp :2 :root :1	tp :4 :root :1
td :1 :user :2	tp :4 :root :1	td :1 :user :2
cours :5 :admin :3	cours :5 :admin :3	cours :5 :admin :3

```
user1@PC:~$ cat file
add 456 ordre_numérique_3 ordre_dictionnaire_4
bac 22 ordre_numérique_1 ordre_dictionnaire_3
car 111 ordre_numérique_2 ordre_dictionnaire_1
dat 1122 ordre_numérique_4 ordre_dictionnaire_2
user1@PC:~$ sort -k 2 file
car 111 ordre_numérique_2 ordre_dictionnaire_1
dat 1122 ordre_numérique_4 ordre_dictionnaire_2
bac 22 ordre_numérique_1 ordre_dictionnaire_3
add 456 ordre_numérique_3 ordre_dictionnaire_4
user1@PC:~$ sort -n -k 2 file
bac 22 ordre_numérique_1 ordre_dictionnaire_3
car 111 ordre_numérique_2 ordre_dictionnaire_1
add 456 ordre_numérique_3 ordre_dictionnaire_4
dat 1122 ordre_numérique_4 ordre_dictionnaire_2
```

❑ Mixing and blending lines : shuf

The command **shuf** allows you to arrange the lines in a random order that differs each time.

Syntax : `shuf file`

Examples :

tp.txt
```
tp :2 :root :1
tp :4 :root :1
td :1 :user :2
cours :5 :admin :3
```

$ shuf tp.txt
```
td :1 :user :2
tp :2 :root :1
tp :4 :root :1
cours :5 :admin :3
```

$ shuf tp.txt
```
cours :5 :admin :3
td :1 :user :2
tp :2 :root :1
tp :4 :root :1
```

❑ Merging lines of files : paste

The paste command groups **n files** into **one**. Concatenates the lines in order (line by line) separated by **tabs** by default.

Syntax : `paste [option] file1 file2 ...`

Examples :

tp1.txt	tp2.txt	$ paste tp1.txt tp2.txt	$ paste -d : tp1.txt tp2.txt
td :user :1	td :usr2 :1	td :user :1 td :usr2 :1	td :user :1 :td :usr2 :1
td :usr1 :3	td :usr3 :3	td :usr1 :3 td :usr3 :3	td :usr1 :3 :td :usr3 :3
tp :root :2	tp :root :2	tp :root :2 tp :root :2	tp :root :2 :tp :root :2
tp :admin :4	tp :admin :4	tp :admin :4 tp :admin :4	tp :admin :4 :tp :admin :4

```
user1@PC:~$ cat test_cut1
1CP
2CP
user1@PC:~$ cat test_cut2
SYS1
ALG2
user1@PC:~$ paste -d":" test_cut1 test_cut2
1CP:SYS1
2CP:ALG2
```

❑ Decomposing files : split

The command **split** allows you to split a file into several pieces of a given size.

Syntax : `split [-l n][-b n[bkm]][fichier[pre]]`

Some interesting options:

- **-l**: split into n lines (except last line)
- **-b**: split to fixed size
- **pre**: generated files will have a name with the prefix **pre**

Example : `split -l 5 liste fich`

Creation of files fichaa, fichab, fichac, ... each containing **5** lines

❑ Comparing two files: cmp

The command **cmp** allows you to check if two files are identical (character-by-character comparison).

Syntax : `cmp file1 file2`

- Identical files ⇨ No output
- Different files ⇨ Indicates the position of the first difference (line and byte)

Example :

tp1.txt
```
td :user :1
td :usr1 :3
tp :root :2
tp :admin :4
```

tp2.txt
```
td :usr2 :1
td :usr3 :3
tp :root :2
tp :admin :4
```

$ cmp tp1.txt tp2.txt
```
tp1.txt tp2.txt differ : octet 6, line 1
```

❑ Difference between two files: diff

The command **diff** specifies the changes you need to make to the two files to make them the same.

Syntax : `diff [-b] file1 file2`

- **-b** : Allows you to ignore spaces
- **file1 file2** : Files to compare

Example :

tp1.txt	tp2.txt	$ diff tp1.txt tp2.txt
user1 :30 :ab	user1 :30 :ab	2c2
user2 :10 :bc	user2 :20 :de	< user2 :10 :bc
user3 :15 :ce	user3 :15 :ce	> user2 :20 :de
user4 :25 :ed	user4 :25 :ed	5d
user5 :31 :df		< user5 :31 :df

❑ Character transformation: tr

The command **tr** (**Translate or Delete Characters**) is used to convert one string to another.

As the command doesn't have a file input argument, it is often used with the **pipe** or **redirection**.

Syntax : `tr [option] string1 string2`

- **-d** : deletion of characters indicated after **-d**
- **-s**: Removed consecutive multiple occurrences of characters indicated after**-s**

Example :

LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/

`tr ["original string"]["new string"]`

To specify input file,

cat [*file path*] | **tr** ["*original string*"]["*new string*"]

tr ["*original string*"]["*new string*"] < [*file path*]

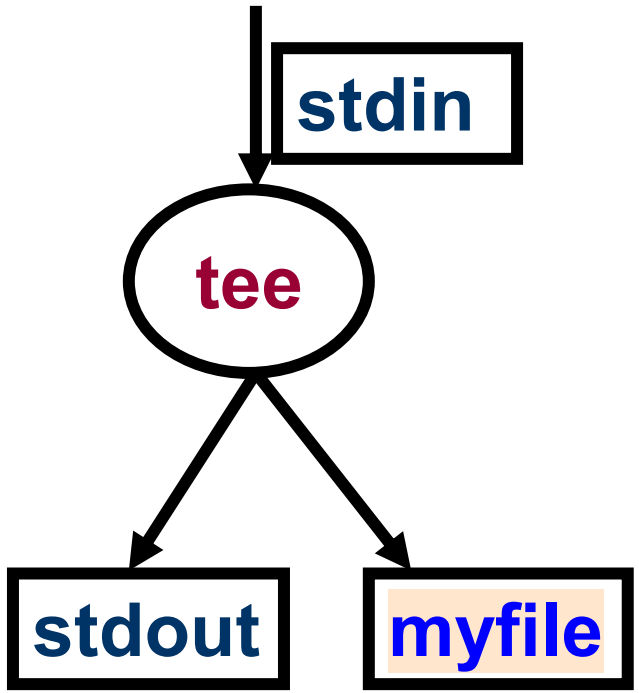
cat tp1.txt
aaaabbbcc cccbbaaa
tr [a-c] [A-C] < tp1.txt
AAAABBBCC CCCBBAAA
tr -d [ac] < tp1.txt
bbb bb

tr -s [bc] < tp1.txt
aaaabc cbaaa
tr [a-c] [x*] < tp1.txt
xxxxxxxxx xxxxxxxx

❑ tee (Replicate the standard output)

The **tee** command reads from standard input and **writes the output to both standard output (the terminal) and one or more files.**

This is useful for **logging** or **debugging** when you want to preserve the output of a command while also viewing it live.



Example :

```
cat readme.txt | tee myfile
```

❑ Grep (Search and Filter Text)

grep is a powerful command-line tool in Linux used to search for **specific patterns** within files.

Syntax :

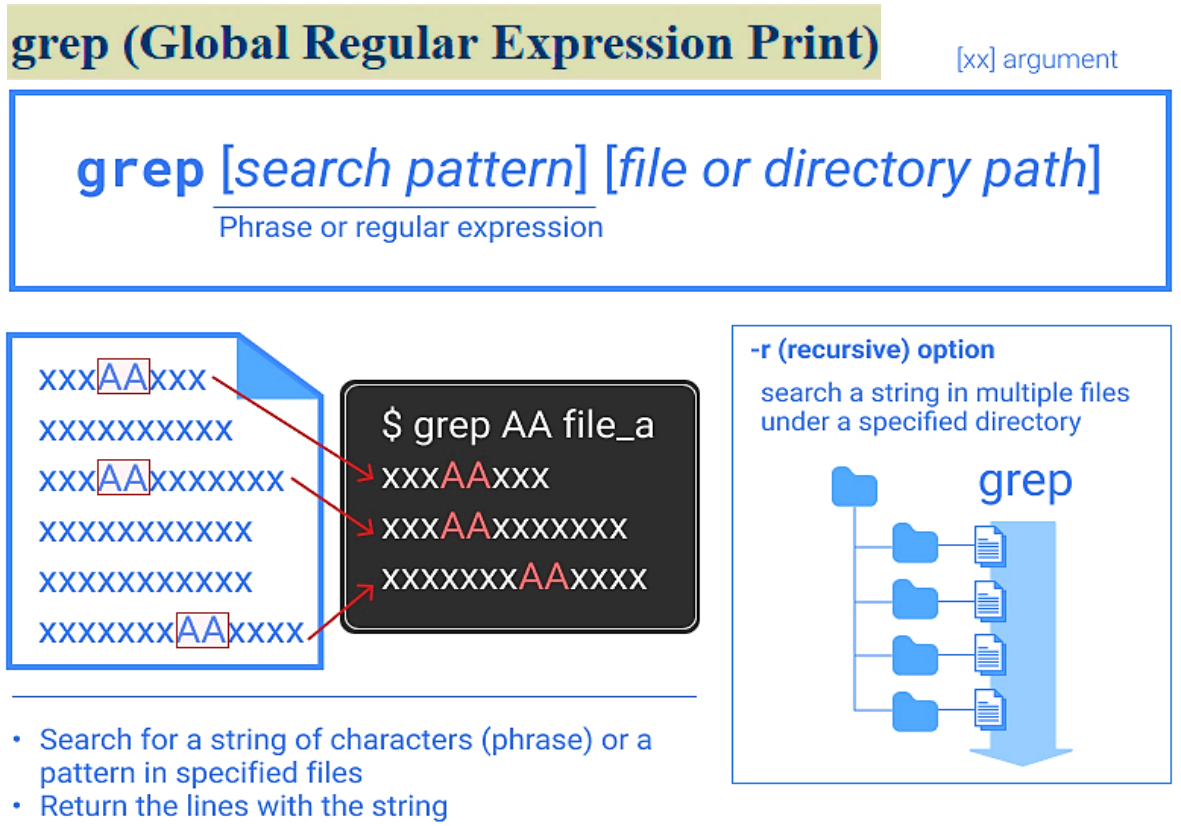
```
grep [options] expression files
```

It can search for text using **regular expressions**, making it highly flexible.

It **filters** and displays lines from a file that match a given pattern, enabling users to find relevant information quickly.

We can also use options like **-i** for case-insensitive searches or **-v** to exclude lines that match the pattern.

LINUX VISUAL GUIDE: Complete beginner's guide
Author: **D-Libro Project** (a lead author: *Ben Bloomfield*)
Edition: First Edition (2024)
D-Libro: https://d-libro.com/course/linux-introduction/



❑ Sed (Stream Editor for Filtering and Transforming Text)

The **sed** command stands for "**stream editor**," and it is commonly used for processing and modifying text in files or streams.

It's a very powerful tool, often used for : ***search-and-replace***, ***inserting***, ***deleting***, or ***modifying*** text in files.

Syntax:

```
sed [options] s/old/new/[g] [files]
```

Some interesting options:

-i : Applies changes to the original files

```
user1@PC:~$ cat test_sed
1CP;SYS1
2CP;ALG2

user1@PC:~$ sed 's/;/:/g' test_sed
1CP:SYS1
2CP:ALG2

user1@PC:~$ cat test_sed
1CP;SYS1
2CP;ALG2

user1@PC:~$ sed -i 's/;/:/g' test_sed
user1@PC:~$ cat test_sed
1CP:SYS1
2CP:ALG2
```

❑ Other important commands

Use the command **man** if you want to get information about the following commands :

- **Awk** – Text Processing and Pattern Matching
- **Head** – Display First N Lines
- **More** – View file content page by page
- **Less** – View file content with more control
- **Tail** – Display Last N Lines
- **Nl** – Number Lines
- **Tac** – Concatenate and Reverse
- **Hexdump** – Display file contents in hexadecimal
- **Aspell** – Spell Checker

THANK YOU for your attention!

Questions ?

NSCS
المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY

مخبر هندسة الأنظمة المعقدة
LABORATOIRE D'INGÉNIERIE DES SYSTÈMES COMPLEXES

For more information about my research works, **Contact Information**:

Dr. Sassi BENTRAD

LISCO Laboratory : http://lisco.univ-annaba.dz/

☎ : +213 …

✉ : sassi.bentrad.enscs@gmail.com // sassi.bentrad@enscs.edu.dz

in : www.linkedin.com/in/sassi-bentrad/

: http://www.bentrad-sassi.sitew.com/

ORCID
Connecting Research and Researchers

: orcid.org/0000-0002-7458-8121

RESEARCHERID : A-9442-2013

SCOPUS Author ID : 44461052600